Python Introduction and Math Operations

1 Purpose

Start using *Python* for basic math operations.

2 Some Creative Commons Reference Sources for This Material

- *Think Python 2nd Edition*, Allen Downey, chapters 1 and 2
- *The Coder's Apprentice*, Pieter Spronck, chapters 3 and 4
- *A Practical Introduction to Python Programming*, Brian Heinold, chapters 1, 3, 10, and 22
- *Algorithmic Problem Solving with Python*, John Schneider, Shira Broschat, and Jess Dahmen, chapters 2 and 3

3 Three Important ***Python*** Object Types

Nearly everything in *Python* is an **object**. *Python* has three object types that are used very frequently; integers, floating point values (floats), and strings. Use `type(x)` to check the type of an object, i.e. `type(42)` or `type("Hello")`

Note: *Only integers and floats can be used to perform math operations.*

- **Integers**: whole numbers with no decimal part
  - `int` type
  - Values like...
    - `0, 1, 2, 3,...`
    - `-1, -2, -3,...`
- **Floating point values (floats)**: non-integer values with a decimal point
  - `float` type
  - Values like...
    - `2.0` or `2.`
    - `-3.45`
- **Strings**: any type of character(s) contained within quotes
  - `str` type
  - Double or single quotation symbols...
    - `"Hello"`
    - `'Python'`
  - Numbers within quotes are strings
    - `"42"`
    - `'2.78'`

Determining object types

```
>>> type(3.14)
<class 'float'>
>>> type(42)
<class 'int'>
>>> type("Python")
<class 'str'>
```

3.1 Converting Types

Examples for each of the following are shown below

- To integer…
  - `int()` function
  - Convert floats - the decimal point and all numbers to the right are dropped
  - Convert strings that contain only an integer value
  - Cannot directly convert a string containing a float to an integer
- To float…
  - `float()` function
  - Convert integers - adds a decimal point and zero
  - Convert strings that contain only a float or an integer value
- To string…
  - `str()` function
  - Convert integers or floats - numeric values are simply enclosed in quotes

Type conversion examples

```
>>> int(3.14159)
3
>>> float(42)
42.0
>>> str(42)
'42'
>>> str(3.14159)
'3.14159'
>>> int("42")
42
>>> float("3.14159")
3.14159
>>> int(float("3.14159"))
3
```

4 Editing and Running Scripts

4.1 Editing Scripts

- *Python* scripts must be written in plain text
  - Stand-alone text editing applications
  - Editors built into a *Python* integrated development environment (IDE)
- Word processors like *Word* should never be used for editing scripts
- *Python* installations have a built-in text editor/IDE named *IDLE*
- Dedicated coding text editors, like *Visual Studio Code* (*VSCode*), often add nice features; including the ability to run scripts directly from the editor
- Good, beginner-friendly IDEs
  - *Mu*
  - *Thonny*

4.2 Naming scripts

- No spaces in the name
- Must start with a letter or an underscore character
- May include numbers
- Must end with `.py` file extension
- Valid script names…
  - `stress_calculator.py`
  - `fibonacci.py`
  - `_calculator.py`
  - `easy_as_123.py`
  - `uscs2si.py`

4.3 Running Scripts

- From *Jupyter*
  - Place the script file in same directory as the *Jupyter* notebook
  - Execute `run script.py` in a code cell using the actual script name, i.e. `run hello.py`
- From a *Python* prompt
  - Change to the directory containing the script file
  - Start *Python*, type `import script_name` at the `>>>` prompt, and press `[return]`
  - For example, `>>> import hello` to run the script `hello.py`
- From a terminal or console command line prompt
  - Type `python` or `python3` followed by the script name with the extension
  - Examples…
    - `python hello.py`
    - `python3 hello.py`

Running a script from a Python prompt

```
>>> import hello
Hello, World!
```

5 The Pythonic Method

There are programming methods and principles that are fairly unique to *Python*. Sometimes the use of these methods and principles is referred to as being **Pythonic**. The development of *Python* as a language has been guided by a number of principles.

Executing `import this` provides a glimpse at the guiding principles.

Zeno of Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

The *Python* developers also have a bit of a sense of humor. They also desire to make *Python* code easy to use and read. Execute the following commands for a little bit of *Python* fun.

Python fun

```
>>> import __hello__
Hello world!

>>> import antigravity   # may not work from a Jupyter notebook

>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
```

Guido von Rossum, the founder and original author of *Python*, was given the title BDFL - Benevolent Dictator For Life - by the *Python* community (although he is now retired from the position). An April Fool's joke regarding the BDFL was published as PEP (Python Enhancement Proposal) 401 in 2009 (https://peps.python.org/pep-0401/) announcing that Guido's title was changed to Benevolent Dictator Emeritus Vacationing Indefinitely from the Language (BDEVIL) and his replacement was Barry Warsaw, who would be referred to as the Friendly Language Uncle For Life (FLUFL). The joke was codified in the next release.

Python fun

```
>>> 2 != 4
True
>>> 2 <> 4
  File "<stdin>", line 1
    2 <> 4
      ^
SyntaxError: invalid syntax

>>> from __future__ import barry_as_FLUFL
>>> 2 != 4
  File "<stdin>", line 1
    2 != 4
      ^
SyntaxError: with Barry as BDFL, use '<>' instead of '!='
>>> 2<>4
True
```

6 Displaying Output Using `print()`

- Use the `print()` function to explicitly display results
- In *Python* or from a *Python* prompt some results may be displayed without using `print()`
- `print()` is the only way to display results within scripts
- The `print()` function displays the results of (nearly) anything included in the parentheses

- Multiple items/objects can be printed with a single `print()` command
- Separate the objects with commas within `print()`
- Spaces are added between objects everywhere there were commas when printed
- Multiple strings in a `print()` function can be connected together (concatenated) by separating them with the `+` symbol

Printing examples

```
>>> print('Hello, World!')
Hello, World!
>>> print("Hello " + "again")
Hello again
>>> print("Python is awesome")
Python is awesome
>>> print('Lumberjacks', "Parrots", 42)
Lumberjacks Parrots 42
>>> print(4 * 5)
20
>>> print(2 * 'Hello')
HelloHello
```

7 Comments

- Use `#` in a *Python* code cell to indicate the start of a comment
- Anything after `#` is ignored by *Python*
- Can start a line with `#`
- Comments can be placed to the right of *Python* commands
- Use comments to describe "why" and not "how" something was done
- A common practice for *Python* scripts is to enclose multiple lines at the top of a script with triple quotes to form a single large comment (see below) called a docstring-style comment

Python comments

```
"""
This is a docstring-style comment
This program does such and such
It was written by Brian Brady
Last revision: 04/15/2022
"""
# This is a single-line comment
x = 42    # Answer to the universe and everything
```

8 Basic Arithmetic

Python uses the following basic arithmetic operators:

- Addition `+`
- Subtraction `-`
- Multiplication `*`
- Division `/`
- Exponentiation `**`

Python adheres to the **PEMDAS** order of operations . . .

1. **P** = Parentheses `( )`
2. **E** = Exponentiation: `4**2` is 4^2
3. **MD** = multiplication and division from left to right: `3*2/3 = 2`
4. **AS** = addition and subtraction from left to right: `5 - 3 + 1 = 3`

Arithmetic operators

```
>>> 67 + 32 - 13
86
>>> 12 * 6 / 2
36.0
>>> 12 / 2 * 6   # same results as previous calculation
36.0
>>> 42 - 10/4
39.5
>>> (42 - 10)/4  # parentheses make a difference
8.0
>>> 10 / 5 + 5
7.0
>>> 10 / (5 + 5)  # use parentheses when needed
1.0
>>> 2**3
8
>>> 4**1/2
2.0
>>> 4**(1/2)
2.0
```

9 Special Division Operators and Functions

9.1 Division Operation in Mathematics

Division involves four different objects: dividend, divisor, quotient, and remainder

- The dividend is divided up by the divisor
  - Dividend ÷ Divisor
  - Dividend / Divisor
- The quotient is the number of times the divisor completely goes into the dividend
- The remainder is what is left (if anything)
- For example, 7/2 results in a quotient of 3 and remainder of 1

9.2 Division in ***Python***

- Normal division `/` always yields a floating point value
- Integer division `//` returns the quotient as an integer
- Also referred to as floor division
- Removes the decimal point and the all digits to the right of it
- Modulo division operator `%` yields just the remainder
- Also referred to as remainder division
- The built-in function `divmod(x, y)` requires two arguments (the dividend and divisor) and returns the quotient and remainder as a pair of values

Division operators for 7/2

```
>>> print('7/2 =', 7/2)
7/2 = 3.5
>>> print('7//2 =', 7//2)
7//2 = 3
>>> print('7%2 =', 7%2)
7%2 = 1
>>> print('divmod(7, 2) =', divmod(7, 2))
divmod(7, 2) = (3, 1)
```

10 Elementary Math Functions and Constants

- *Python* is a very extensible programming language
- Modules (libraries) are available for nearly everything
- Need to `import` modules whose commands you want to use
- `import` the `math` module to use math functions beyond basic arithmetic
- `dir(math)` or `print(dir(math))` to see available `math` commands

Importing math and listing the available math commands

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
 ↪  'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb',
 ↪  'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc',
 ↪  'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
 ↪  'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
 ↪  'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10',
 ↪  'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod',
 ↪  'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
 ↪  'trunc']
```

- Specific functions/commands can be imported from the `math` (or any other) module (see example below)
- Do not have to include `math.` for commands that are specifically imported

Importing specific math functions

```
>>> from math import cos, sin, tan, pi
>>> cos(pi/3)
0.5000000000000001
```

- You can also import all commands from a module by using the `*` wildcard (as shown below).
- This is not considered good form nor is it very Pythonic and is frowned upon in most cases
- Please do not use this method of importing modules unless specified by the instructor

Not a good practice, but it works

```
>>> from math import *
>>> sin(pi/3)
0.8660254037844386
```

11 General Math Functions

- `abs(x)` $= |x|$ (absolute value is not part of the `math` module)
- `math.sqrt(x)` $= \sqrt{x}$ (x must not be negative)
- `math.exp(x)` = `math.e**x` $= e^x$
- `math.log(x)` $= \log x$ or $\ln x$ (this is the natural log)
- `math.log(x, b)` $= \log_b x$ (the logrithm of base b)
- `math.log10(x)`$= \log_{10} x$
- `math.factorial(x)` $= x!$ (x must be a positive integer)

General math functions

```
>>> abs(-100)
100
>>> math.sqrt(5)
2.23606797749979
>>> math.exp(2)
7.38905609893065
>>> math.log(math.exp(2))
2.0
>>> math.log(25, 5)
2.0
>>> math.log10(1000)
3.0
>>> math.factorial(5)
120
```

11.1 Math Constants

The following commands are used for the mathematical constants π, e (Euler's number), ∞, and "not a number"

- `math.pi` $= \pi$
- `math.tau` $= 2\pi$
- `math.e` $= e$
- `math.inf` $= \infty$
- `math.nan` = "Not a number"

11.2 Trigonometric Functions

- The `math` module includes full support for trigonometric functions
- Standard trig functions require/use angle units of radians, not degrees
- Convert angles from degrees to radians using `math.radians(x)` (described later)

Trig functions

```
>>> math.sin(math.pi/4)
0.7071067811865475
>>> math.cos(math.pi/3)
0.5000000000000001
>>> math.tan(math.pi/4)
0.9999999999999999
```

- Inverse (arcus) trig functions return angles in radians, not degrees
- There is a second inverse tangent function `atan2()`

  - Accepts two arguments `y, x` instead of one
  - Returns a quadrant-specific angle based on an x and a y value
  - Arguments are in y, x order to match y/x, i.e. `atan2(y, x)` equals $\tan^{-1}(y/x)$

Inverse trig functions

```
>>> math.asin(0.9)
1.1197695149986342
>>> math.acos(0.6)
0.9272952180016123
>>> math.atan(1.0)
0.7853981633974483
>>> math.atan2(-2, -4)
-2.677945044588987
```

- Angles can be converted from degrees to radians or from radians to degrees
  - `math.degrees()` converts an angle from radians to degrees
  - `math.degrees(math.pi/2)` converts $\pi/2$ to degrees
  - `math.radians()` converts an angle from degrees to radians
  - `math.radians(30)` converts 30° to radians

Trig functions with angle conversions

```
>>> math.sin(math.radians(30))
0.49999999999999994
>>> math.cos(math.radians(45))
0.7071067811865476
>>> math.degrees(math.atan(1.0))
45.0
>>> math.degrees(math.asin(3/5))
36.86989764584402
```

11.3 Rounding Related Functions

- `round(x, n)`
  - Rounds values towards zero if the decimal part is less than 0.5 and away from zero otherwise
  - Second argument is optional and is used to set the number of decimal places for rounding
  - `round(3.14159, 2)` will round to 2-decimal places, resulting in `3.14`
  - `round(8675309, -2)` will round to the hundreds place, resulting in `8675300`
- `math.trunc(x)`
  - Always drops off the decimal portion
  - Leaves an integer value
- `math.ceil(x)`

    - Returns the next integer value towards positive infinity if the argument has a decimal part
- `math.floor(x)`
    - Returns the next integer value towards negative infinity if there is a decimal part

Rounding related functions

```
>>> round(math.e)
3
>>> round(math.e, 3)
2.718
>>> round(8675309, -2)
8675300
>>> math.trunc(math.pi)
3
>>> int(math.pi)  # same as use math.trunc()
3
>>> math.ceil(2 * math.pi)
7
>>> math.ceil(2.0001)
3
>>> math.floor(44.99)
44
```

12 Using Names (Variables)

- Can assign names (variables) to almost any value or object
- Use the variables in place of the value or object
- Case-sensitive, i.e. `a` and `A` are not the same
- Variable naming...
    - Must start with a letter or underscore
    - Can also contain numbers and underscores
    - No spaces and other special characters
    - Restricted *Python* names (keywords) cannot be used as variable names
    - Don't use built-in functions or commands as variable names
        - For example, don't use `abs = 100` or `math.pi = 3`
        - Will overwrite the functions `abs()` and `math.pi`
- Using `del variable_name` will clear `variable_name`
- Use descriptive variable names like shown below

Assigning names (variables)

```
>>> answer_to_universe = 42
>>> print(answer_to_universe)
42
>>> radius = 10
>>> print('radius =', radius)
radius = 10
>>> bad_guy = "Darth Vader"
>>> total_time = 30
```

- *Python* does not display any results when a value or calculation is assigned to a name (see above)
- Use the `print()` function in order to explicitly display such results

13 Special Commands

- Last calculated result not assigned to a name is accessed using the underscore `_` character
  - This is not normally done within a script or a function
  - Acts like the "ans" key on a calculator
- `dir()` returns a list of all current names and modules in memory
- `dir(module)` will return the available commands, functions, and methods in a module named `module`
- *Jupyter* and *iPython* have some *magic* commands
  - `who` is like `dir()` but formatted differently
  - `whos` is like `who` but it also includes types and value information